.NET Full Stack Development Program

—

Day 12 ASP.NET Core MVC

Outline

- MVC Introduction
  - Controller
    - Action
    - Routing
  - View
    - Data passing from Controller to View

MVC

MVC is an architecture that separates business logic, presentation and data. In MVC,

- M stands for Model
- V stands for View
- C stands for controller.

MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

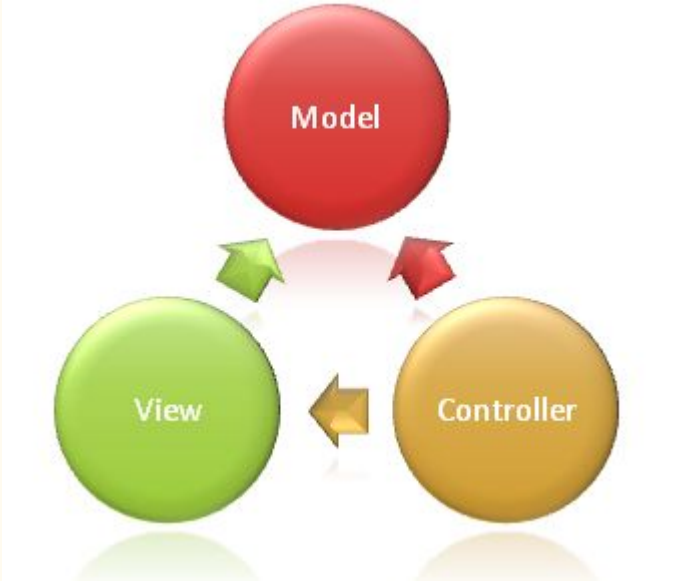
MVC

- Model
  - The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it
- View
  - Views are responsible for presenting content through the user interface.
- Controller
  - Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render

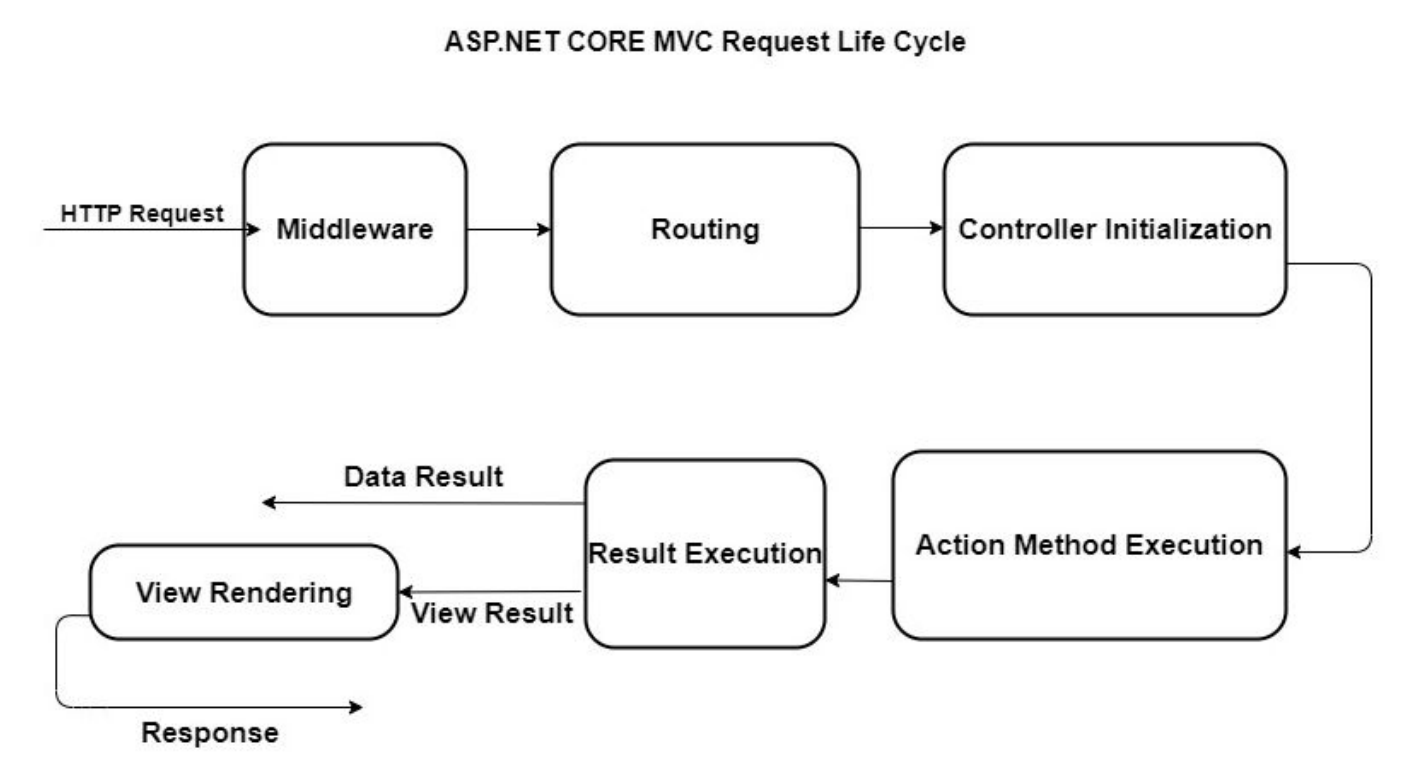
Why MVC

Separation of Concern

- Loose coupling between model, view, and controller
- Easy to scale



ASP.NET Core MVC Request Life Cycle



Controller

- A **controller** is used to define and group a set of actions.
- An **action** (or action method) is a method on a controller which handles requests.
- Controllers logically group **similar actions together**.
- Requests are mapped to actions through **routing**.

Controller

- A controller is an instantiable class, usually public, in which at least one of the following conditions is true
  - The class name is suffixed with **Controller.**
  - The class **inherits** from a class whose name is suffixed with **Controller.**
  - The **[Controller]** attribute is applied to the class.

Action

- Public methods on a controller, except those with the **[NonAction]** attribute, are actions. Parameters on actions are bound to request data and are validated using model binding.
- Actions can **return anything**, but frequently return an instance of **IActionResult** (or Task<IActionResult> for async methods) that produces a response.

Action

Anything that could be returned

- HTTP Status Code
- Redirect
- View
- Formatted Response

Routing

Routing is the process through which the application **matches** an incoming URL path and executes the corresponding action methods. ASP.NET Core MVC uses a routing middleware to match the URLs of incoming requests and map them to specific action methods.

```
app.UseRouting();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Routing

There are two types of routing for action methods:

- Conventional Routing
- Attribute Routing

Conventional Routing

When we create a new ASP.NET Core MVC application using the default template, the application configures a default routing.

```
app.UseRouting();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

*You can define **multiple conventions** or to add routes that are dedicated to a specific action

Attribute Routing

By placing a route on the controller or the action method, we can make use of the Attribute Routing feature.

- **Route()**
- **Http[Verb]()**

```
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

```
[Route("")]
[Route("index")]
0 references
public IActionResult Index()
{
    _dbContext.Persons.Add(new Person() { firstName = "mm", lastName = "ll" });
    _dbContext.SaveChanges();
    return View();
}

[HttpGet("welcome/{name}")]
0 references
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}

[Route("privacy")]
0 references
public IActionResult Privacy()
{
    return View();
}
```

View

The **view** handles the app's data presentation and user interaction. A **view** is an HTML template with embedded Razor markup. **Razor markup** is code that interacts with HTML markup to produce a webpage that's sent to the client.

- Views that are specific to a controller are created in the Views/[ControllerName] folder.
- Views that are shared among controllers are placed in the Views/Shared folder.

View

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor Markup

- Razor markup starts with the @ symbol.
- Run C# statements by placing C# code within Razor code blocks set off by curly braces ({ ... }).
  - @{...}

Pass Data from Controller to View

Pass data to views using several approaches:

- Strongly typed data: **viewmodel**
- Weakly typed data
  - **ViewData** ([ViewData] Attribute)
  - **ViewBag**

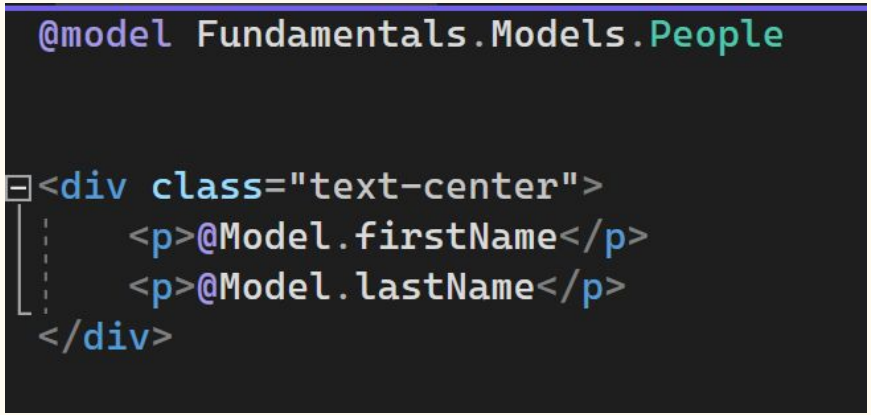
Strongly-typed data (viewmodel)

Specify a model type in the view. This model is commonly referred to as a **viewmodel**. You pass an instance of the **viewmodel** type to the view from the action

- Using a viewmodel to pass data to a view allows the view to take advantage of **strong type checking**. **Strong typing** (or strongly typed) means that every variable and constant has an explicitly defined type (for example, string, int, or DateTime). The validity of types used in a view is checked at compile time.

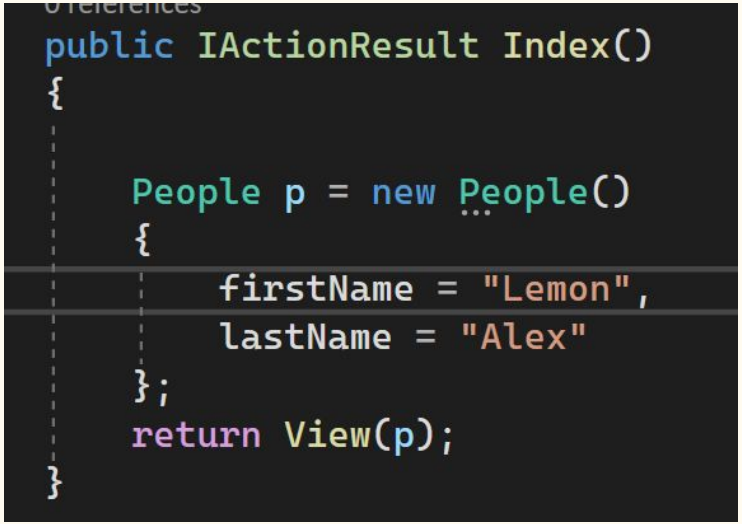
Strongly-typed data (viewmodel)

- Specify a model using the **@model** directive. Use the model with **@Model:**



Strongly-typed data (viewmodel)

- To provide the model to the view, the controller passes it as a parameter:



*There are no restrictions on the model types that you can provide to a view. We recommend using **Plain Old CLR Object (POCO)** viewmodels with little or no behavior (methods) defined.

Weakly Typed Data

Weak types (or loose types) means that you don't explicitly declare the type of data you're using. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views

This collection can be referenced through either the **ViewData** or **ViewBag** properties on controllers and views.

- The **ViewData** property is a dictionary of weakly typed objects. T
- The **ViewBag** property is a wrapper around ViewData that provides dynamic properties for the underlying ViewData collection.

Weakly Typed Data (ViewData)

ViewData is a **ViewDataDictionary** object accessed through **string keys**

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

Weakly Typed Data (ViewData)

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

Weakly Typed Data ([ViewData] attribute)

```
public class HomeController : Controller
{
    [ViewData]
    public string Title { get; set; }

    public IActionResult About()
    {
        Title = "About Us";
        ViewData["Message"] = "Your application description page.";

        return View();
    }
}
```

```
<title>@ViewData["Title"] - WebApplication</title>
```

Weakly Typed Data (ViewBag)

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>
```

ViewData vs. ViewBag

- `ViewData`
  - Derives from ViewDataDictionary, so it has dictionary properties that can be useful, such as `ContainsKey`, `Add`, `Remove`, and `Clear`.
  - Keys in the dictionary are strings, so whitespace is allowed. Example: `ViewData["Some Key With Whitespace"]`
  - Any type other than a `string` must be cast in the view to use `ViewData`.
- `ViewBag`
  - Derives from `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData`, so it allows the creation of dynamic properties using dot notation (`@ViewBag.SomeKey = <value or object>`), and no casting is required. The syntax of `ViewBag` makes it quicker to add to controllers and views.
  - Simpler to check for null values. Example: `@ViewBag.Person?.Name`

Question?